Matrix Scheduler Reloaded

Peter G. Sassone Jeff Rupley II Edward Brekelbaum Gabriel H. Loh† Bryan Black

Intel Microarchitecture Research Lab (MRL), Austin TX
†College of Computing, Georgia Inst of Technology, Atlanta GA
peter.g.sassone@intel.com

ABSTRACT

From multiprocessor scale-up to cache sizes to the number of reorder-buffer entries, microarchitects wish to reap the benefits of more computing resources while staying within power and latency bounds. This tension is quite evident in schedulers, which need to be large and single-cycle for maximum performance on out-of-order cores. In this work we present two straightforward modifications to a matrix scheduler implementation which greatly strengthen its scalability. Both are based on the simple observation that the wakeup and picker matrices are sparse, even at small sizes; thus small indirection tables can be used to greatly reduce their width and latency. This technique can be used to create quicker iso-performance schedulers (17-58% reduced critical path) or larger iso-timing schedulers (7-26% IPC increase). Importantly, the power and area requirements of the additional hardware are likely offset by the greatly reduced matrix sizes and subsuming the functionality of the power-hungry allocation CAMs.

Categories and Subject Descriptors. C.1.0 [**Processor Architectures**]: [Single Data Stream Architectures]

General Terms. Algorithms, Performance, Design

Keywords. Microarchitecture, Matrix, Scheduler, Wakeup, Picker

1. INTRODUCTION

To the consternation of microarchitects, process scaling has provided the vast majority of the speedup seen in processors over the last 40 years. In fact, microarchitecture is increasingly hampered by process constraints such as the relative growth of wire delays each generation. Though the maximum speeds can be tweaked through sizing and repeaters, wires still limit the dimension of many structures in a modern processor design. A quintessential example is the scheduler. Increasing the selection of instructions which can issue on a given cycle is an effective method for increasing performance in a simulator — moving from a 16-entry to a 64-entry scheduler creates a speedup of 39% in our simulator. Yet, the physical design of such large schedulers continues to be prohibitive for real commercial designs — the latest Intel and AMD desktop/server cores only have integer scheduler sizes of 24 to 32 entries [1, 17] primarily for latency reasons. The fundamental issue is that the delay of the scheduler loop is proportional (or worse) to the number of entries. This is true on both sides of a traditional scheduler: the wakeup side responsible for dataflow ordering, and the picker side responsible for resource allocation and age tracking.

To address scalability on the wakeup side, we present a straightforward modification to wakeup matrices to enable much larger and/or faster instruction windows than traditional methods. The key idea is the subscription of wakeup matrix columns (broadcast-to-wakeup communication channels) by consuming operations only on-demand. Instead of supporting the maximum number of operands to be waited on — a worst-case assumption which condemns traditional schedulers to poor scalability, we can support a small number and still achieve excellent performance. In our experiments, we only need to track 12 to 16 broadcasts for typical scheduler sizes, and only about 20 for very large schedulers.

Similarly for the picker side, we introduce an indirection technique which greatly reduces the picker matrix size to cover the same issue window. The key observation here is that the picker's primary complexity, maintaining all-to-all ordering resolution, is a significant over-design. Very similar performance can be achieved by tracking groups of 12 instructions rather than every instruction. This ordering approximation can reduce the size of the picker matrix by 60-90%, shortening the critical path latency significantly.

Together these techniques produce a new type of matrix scheduler which is far more efficient than traditional square designs. For architects wishing to hold IPC constant, our estimates show that the combined techniques can reduce the schedule-loop delay by 17-58% over a traditional matrix holding the same number of instructions. If we hold delay constant from a traditional design, the thin matrix design can capture a larger issue window, increasing IPC by 7-26%.

This paper is organized as follows. In Section 2 we discuss the wakeup side of the scheduler, including background, related work, and our proposal for increasing the efficiency of the matrix structure. Section 3 follows the same outline for the picker side. Section 4 discusses simulation methodology and then evaluates both enhancements separately and combined. Section 5 concludes with discussion of on-going work.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

2. WAKEUP

In this section we address the first half of a traditional scheduler, the wakeup logic. Before we discuss our proposal for increasing the efficiency of a wakeup matrix by allocating columns conservatively, we briefly review background on the functionality of the wakeup logic and the root of the scalability issues.

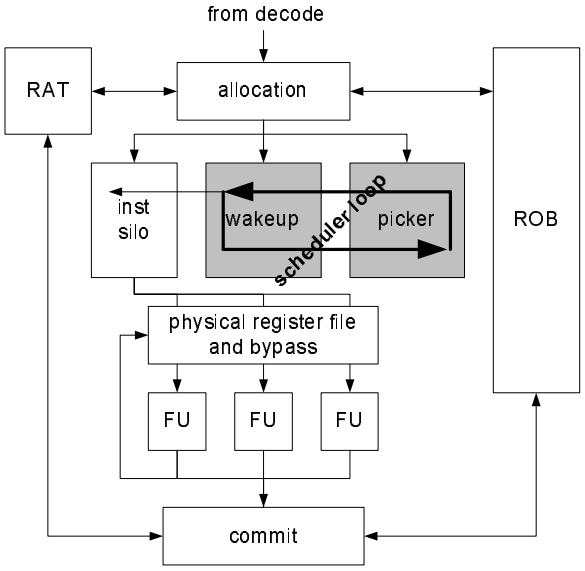

Figure 1: The wakeup and picker portions of the scheduler and their relation to a typical out-of-order core.

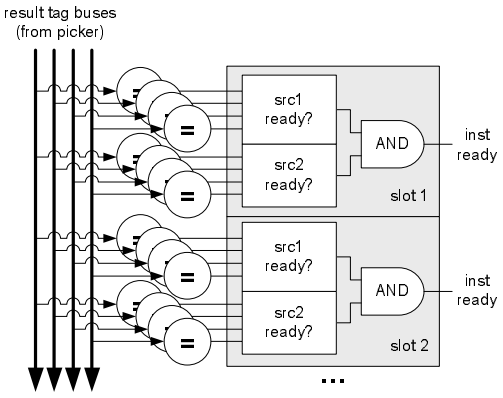

Figure 2: Illustration of a CAM-based scheduler with the multitude of needed comparators.

2.1 Background

The wakeup portion of the scheduler is very similar to the wait-match unit of a dataflow machine [20]. The purpose is to observe results being generated in order to identify instructions that are now ready for execution. Figure 1 shows the relation of wakeup to the rest of a conventional processor core. The dependency cycle formed between the wakeup logic which identifies ready instructions, and the picker logic which selects a set of ready instructions for execution, forms a tight loop which is well known as critical to performance [2].

There are two conventional methods to implement the matching algorithm in modern schedulers: (a) content addressable memories (CAMs) and (b) dependency matrices. Despite its disadvantages, content addressable memories are how most commercial schedulers are implemented. The result tags (typically just the physical register number) of selected instructions are broadcasted on a set of result buses, one for each functional unit with a writeback port. Each set of result buses is connected to comparators (XNOR gates) at each entry to allow instructions to match their sources against those being generated. When both of the instruction's sources are ready, the

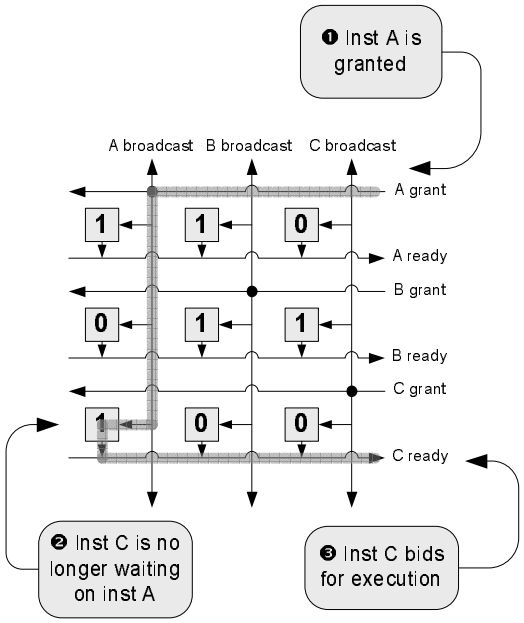

Figure 3: Illustration of matrix wakeup with the critical path shaded.

instruction as a whole is ready and bids for execution. An illustration of this hardware is shown in Figure 2.

Unfortunately, the most basic implementation of a CAM-based scheduler requires one comparator for each source on each write-back port. Thus a scheduler for 32 two-source instructions on a machine with four writeback ports needs a whopping 32x2x4=256 comparators! The frequent switching of these comparators plus the long lengths and load capacitance of the result buses creates clear power and timing concerns.

In response to these concerns, a dependency matrix was proposed to implement this wait-match function instead [14]. An illustration of this approach is shown in Figure 3. This wakeup matrix has one row and one column for every instruction in the scheduler. Each cell holds one bit of state representing whether the instruction assigned to this row is waiting on a dependency from the instruction assigned to this column. These wakeup vectors are set via allocation comparators which put a 1 in the vector if the corresponding column will be producing this result tag. These comparators are unfortunately quite cumbersome, both in area and power, given the throughput of comparisons that must be made per cycle. As instructions become ready, they clear their respective column (i.e., set all bits in the column to 0). When a row is entirely clear of dependencies (all 0), a wired-or across the row produces a ready bit for the instruction as a whole. These row and column-based functions are faster and lower power to implement than CAM-based matching, but the N^2 nature of the matrix makes larger schedulers still difficult to implement.

Academic researchers have proposed dataflow prescheduling [5, 8, 21, 22] and dependence collapsing [3, 24, 25], both of which use dataflow information to reduce or eliminate the need for wait-match in the schedule loop. However, both techniques require complex power-hungry analysis of the program to work. This analysis can be in the processor front-end which will likely increase the branch penalty, or in the back-end which will likely require additional metadata storage. An interesting approach applicable only to CAM-based schedulers is tag elimination [7, 16]. These researchers

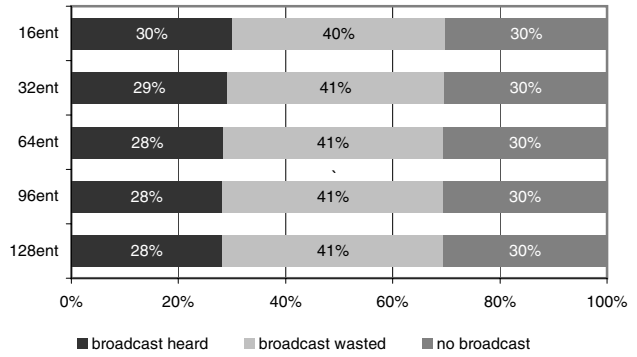

Figure 4: Distribution of tag broadcasts across scheduler sizes.

observe that the worst-case design of two comparators per port per instruction is overkill — only 10-20% of instructions require two source wakeups for most benchmarks. As such, a design which manages an average-case number of comparators is more efficient and sacrifices only minor slowdown. Unfortunately, the nature of these approaches are specific to CAM-based wakeup, which is inherently slower and less efficient than matrix wakeup even with these enhancements [14].

2.2 Wasted Broadcasts

Despite the speed of matrices, they have incredibly low information density. A typical snapshot of a wakeup matrix during execution shows very few dependenices represented — commonly only 10-20. Indeed, this confirms the intuition behind tag elimination: the number of live sources in the scheduler is quite low at any given time. Unlike in a CAM-based scheduler, however, sources in a wakeup matrix are physically free. Removing a subset of matrix intersections (source matches) might reduce power slightly but does not change the lengths of the dominant ready, broadcast, and grant lines. On the other hand, reducing the number of broadcast channels (columns) greatly reduces the width of the matrix and these wires.

If previous authors observed that few wakeup tag comparators are needed for near-ideal performance, it is likely that few tag broadcasts are also needed. To test this theory, we run our simulator across our benchmark suite (simulation and benchmark details in Section 4) and classify all scheduled dynamic instructions into three categories. The first category is *broadcast heard*. This is the textbook case where the instruction generates a broadcast and there is at least one consumer in the scheduler which is listening for it. The second category is *broadcast wasted* where the instruction generates a broadcast, but there are no consumers in the scheduler. A consumer might eventually arrive in the scheduler, but it will be told during allocation that this value is waiting in physical register storage; the broadcast here is wasted. The final category is *no broadcast*, which means the instruction does not generate a renamed destination. This could be a branch, a store, or a control instruction whose result is not renamed.

We plot the distribution of these broadcast states for various scheduler sizes in Figure 4. The latter two categories, *wasted broadcast* and *no broadcast*, combine for a total of 70-71% across all scheduler sizes, confirming our hypothesis that most instructions do not need the functionality of tag broadcast regardless of the scheduler size. Combined with the fact that big schedulers are rarely full of producers (often still refilling from a pipeline flush), we can see that the number of broadcast-to-wakeup communication channels needed at any given time is low.

2.3 Hardware Modifications

In order to exploit the scarcity of needed result tag broadcasts, we modify the allocation and wakeup portions of the microarchitecture to support tag broadcast on only a subset of the scheduler entries. Figure 5 shows the hardware changes we implemented for wakeup subscriptions, with new hardware shaded in gray. We begin with a wakeup matrix as our baseline design. A traditional matrix, as discussed in the previous section, supports all-to-all broadcast by having as many columns (broadcast channels) as it has rows (ready generation channels). Our matrix, however, need not support total broadcast so we will construct it with fewer columns than the maximum. Our later results will show this width need only be around 12-16 to show very favorable performance, even for high-capacity schedulers. To manage the subscription of columns, we also add a small table called the Wakeup Allocation Table (WAT) which maps architectural registers to column numbers. This table is accessed in parallel to the Rename Alias Table (RAT).

Each WAT entry can be in one of three states. First is *unallocated*, which indicates the WAT entry data field is a pointer to the scheduler entry of the last instruction which produced this register. For instance, an instruction sourcing R4 might see the WAT entry for R4 as "unallocated, 20", meaning that this register will be produced by the instruction at scheduler entry 20. As this consumer instruction will need to observe when the result for R4 is ready, we establish a communication channel between the producer and consumer, i.e. a matrix column. Thus the allocation logic requests a column number from the Wakeup Free List, a structure similar to other microarchitectural free-lists, which holds unallocated wakeup columns. This column number is assigned to the consumer by setting the appropriate bit in the dependency vector. The column number is also sent to the producer instruction. Thus, in our example, the producer at entry 20 will be told to broadcast (raise the broadcast line) on this column. This column is then assigned to this WAT entry (R4 in our example), and the state is changed to *allocated*. We have now subscribed this register to a column.

The *allocated* state means this architectural register is currently mapped to a matrix column, and the data field in the WAT indicates that column number. In our example, if a subsequent consumer of R4 allocates, the WAT lookup might return "allocated, 3", which means this register is assigned to matrix column 3. The column number is read and used to set this instruction's dependency vector appropriately. As the producer already knows to broadcast on this column, no notification of the producer is needed in this condition.

When the producer is eventually granted execution, it will fire a broadcast along column 3 and wakeup will proceed as in a traditional matrix. We tell the WAT that this register is now ready so that future consumers do not set their dependency vectors or notify a producer. We do this by changing the WAT state of the destination register to the third and final state, *ready*. We must remember, however, that the WAT is indexed by architectural register and is thus updated like a future file [28]. Sequence numbers, shift registers, or a similar system are used to restrict WAT updates to the last-allocated writer; otherwise an older instruction could overwrite newer information about the state of R4.

The wakeup column is freed if one or both of the following two conditions is met: (1) the producer of the column has left the scheduler and/or (2) all of the consumers of the column have left the scheduler. The second condition is an optimization for when a pipeline flush occurs between a register's producer and consumer(s). Even if the producer remains in the scheduler, we can safely revert the register state to *deallocated* and return the column to the free list. Regardless of why the column is freed, a pointer to it is returned to the Wakeup Free List for future use.

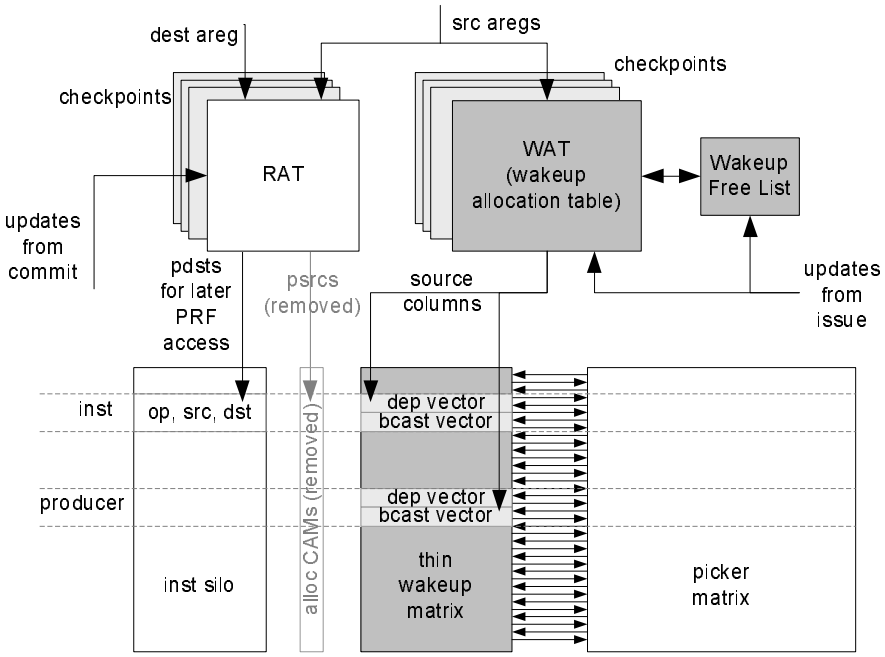

Figure 5: Hardware modifications needed for matrix columns subscriptions (not to scale).

The reader should note that we do not free the column when the producer broadcasts on it. Though this would be safe in many machines, the use of speculative scheduling in our processor model makes this hazardous. Speculative scheduling means that consumers of a load are granted execution assuming the load will hit in the L1 data cache. If the load actually misses, the consumer will be reset in the scheduler and await a second broadcast from the load when it hits. Thus one instruction can broadcast multiple times on the same column, so we cannot assume finality after the first broadcast. Only after the producer leaves the scheduler can we be assured that no further broadcast of this column will take place.

The primary goal of our proposal is to shorten the critical path through the wakeup matrix. Figure 6 shows the necessary changes, with new hardware shaded in gray. The reader should note this cell is designed using active-low dynamic logic, as is common with high-speed commercial implementations. The specific design of the matrix cell is not particularly relevant, however, as there are other reasonable ways of achieving the same functionality. Our purpose is to simply show a working example, and how minor the changes to the wakeup matrix cell are.

In our baseline example, picker grants enter from the right, which become vertical broadcast signals at the transpose column (row equals column). The broadcast bit clears any dependencies in this column, which is stored in a 1-bit SRAM cell labeled *datadep*. If the cell still has a dependency, it will pull down the *row-ready* signal, indicating this instruction is not ready to bid yet. If no instructions pull down *row-ready*, the line will stay high and the picker will interpret that signal as a bid. We will discuss the picker side of this circuit loop in the next section.

To enable our column subscription technique, we modify the cell with an additional bit of information, *writecol*, which indicates if this row (producer) is assigned to broadcast on this column. The *writecol* bit is assigned by the WAT during allocation of the first consumer, and it is guaranteed that there is only one bit set per row and per column. This bit is now ANDed with the existing grant input to determine the broadcast this cycle. This adds one gate delay to the ready output path of this cell, but as all cells compute their ready in parallel, the total row-ready critical path is only increased by one gate. We believe this minor impact is strongly outweighed by the reduction in critical wiring path through the thinner matrix.

2.4 Hardware Considerations

There are many considerations in our approach deserving brief discussion. Most important is that an instruction's dependency vector is now set entirely by the WAT. This subsumes the functionality of the allocation comparators in a traditional wakeup matrix. Though the WAT is heavily ported and the accompanying logic is not negligible, the removal of these comparators should go a long way toward offsetting the power and area costs.

Also of interest is that producer notification of broadcast columns can create a "ships passing in the night" scenario. If the producer is granted execution at the same time it is assigned a broadcast column, care must be taken that the dependency vector of the consumer is set properly. Otherwise deadlock could occur as the consumer waits for a column broadcast that will never come. Ships-passing logic is not intractable, however, and it is handled cleanly for other purposes in many existing scheduler designs.

Finally, we should note that the WAT is, by its very nature, a speculative structure. It will invariably track mappings down the wrong path and will need to be restored to a correct state. In Figure 5 we show checkpoints as a possible solution, but there are numerous equally valid methods for state restoration. A detailed analysis of all solutions is beyond the scope of this work, but the most obvious solution is to use the same method as used for the RAT. Along those lines, it is conceivable that the WAT information could simply be incorporated into the RAT. Both the RAT and WAT are indexed by architectural register number and are accessed during the same phase of allocation. This merging is not ideal, however, since WAT reads are only performed for instruction's sources, and the WAT will be updated at both execute and dispatch. Additionally, we feel it is generally simpler to architect new structures than modify existing verified structures.

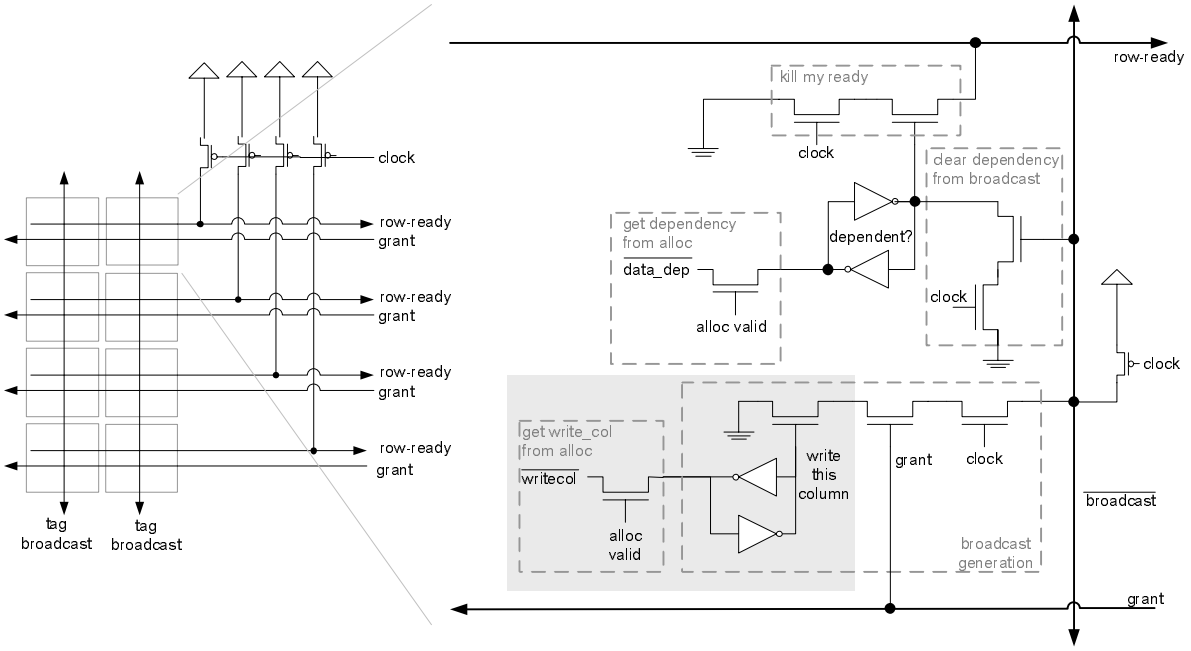

Figure 6: Example 4x2 wakeup matrix (left) and matrix cell circuit (right) with new circuits shaded in grey.

3. PICKER

The other half of the schedule loop is the picker, sometimes called the select logic. Figure 1 shows the relation of the picker to the wakeup logic and the rest of the out-of-order pipeline. As with the previous section, we will first briefly review pickers and their scalability issues before discussing our modifications.

3.1 Background

At its most basic level, the picker is an arbitrator which determines which instructions are permitted to dispatch to which execution resources. The pickers in modern microprocessors have several different factors to weigh in their selection process:

Instruction Readiness. Not all instructions in the scheduler are arbitrating for execution — only those that have been notified via the wakeup logic that their inputs are ready offer themselves to the picker via a bid signal. In the worst case, however, all instructions in the scheduler might be ready and bidding.[1]

Candidate Resource List. This is a list of possible execution-resources for an instruction type. For instance, *add* instructions might be executable on three different integer ALUs, and *fpdiv* instructions might only be executable on the single floating-point divider. The picker must assure that instructions are only dispatched to appropriate execution units.

Resource Availability. The picker must also understand execution resource availability and avoid conflicts. This is especially tricky for resources with variable latencies, such as a load unit or divider. Additionally, the picker must often understand the availability of (often complex) bypass paths and physical register read ports.

Conflict Resolution Data. This category includes all the information provided by the bidding instructions to resolve resource conflicts. For instance, if there is only one FP multiplier and there are two FP multiplies ready to execute, the picker can use additional information from the instruction (such as its age or priority) to decide which is granted and which must wait.

In a traditional picker, all of this information is combined to form a set of grant signals, which are used to communicate which of the bidding instructions are selected for execution and on which resources. These granted instructions then broadcast their destination register tags to the other entries in the scheduler. When an instruction has heard all of its sources broadcasted (the ones that weren't already ready upon entering the scheduler), it can safely raise its bid line to the picker and the scheduler loop is closed.

The most complicated aspect of the picker is the conflict resolution, most often done with age as the tie breaker — older instructions go before younger ones. Other restrictions (i.e., resource conflicts) can generally be handled by fast cancellation logic which easily finish "underneath" the age comparison. A straightforward way to perform age ordering is an age matrix, shown in Figure 7. The age matrix has one row and one column per instruction in the scheduler, where the column number equals the row number. The cell holds one bit which represents an age conflict bit; it is set to 1 if the instruction at this row is older than the instruction corresponding to this column, 0 if the row instruction is younger. An instruction allocates with all conflict bits set (it is younger than every other instruction). As other instructions enter the scheduler, the corresponding columns are zeroed indicating that they are now older than the instructions using those columns. Superscalar allocate follows the same algorithm, but shortcuts might be taken in setting the age-bits appropriately between the instructions in the allocate group. It should also be noted the bits along the diagonal are not used since an instruction cannot be older or younger than itself.

Figure 7 also shows an example of resolution between instructions A and C. Bids from these instructions enter from the left and by

[1] In reality, one entry would contain the producer instruction which readied all the others, so not *all* entries can bid at once.

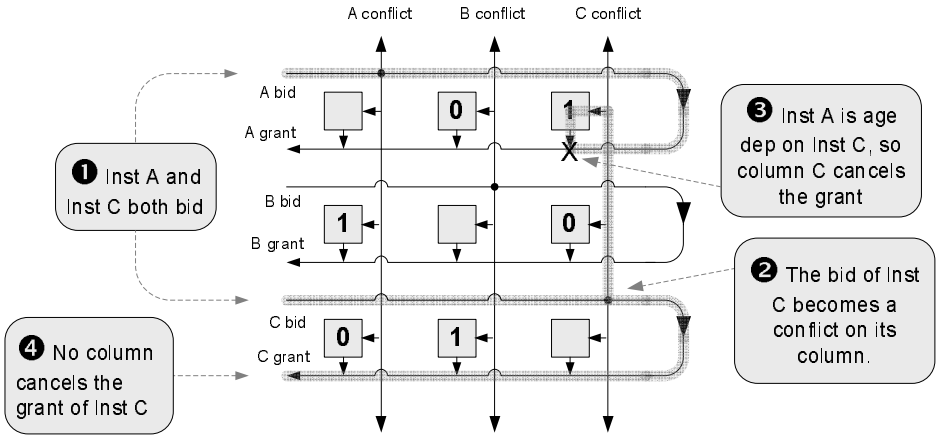

Figure 7: An illustration of an age matrix with an example conflict resolution between younger instruction A and older C.

default result in a grant back to that instruction. This is illustrated with the loop-back from bid to grant at the far right of each row. Each cell in the row, though, must cancel the outgoing grant if two conditions are met. The first is that another bidding instruction broadcasts a conflict. This conflict is directly connected to the bid on the transpose row. So if the instruction on row 9 bids, there will be a conflict broadcast on column 9. The other condition is that the age-conflict bit in this cell is set. Thus if the other instruction bids and it is older, the grant for the current instruction is canceled. If no conflicts occur, the tentative grant will continue unimpeded out of the age matrix and become an actual grant to the instruction on this row. The logic and wiring is slightly more complicated when multiple execution resources are being arbitrated, but the basic design holds.

The scalability of the age matrix, however, is limited. Its size grows quadratically as more scheduler entries are added, and the critical path, where the bottom-most instruction must cancel the top-most instruction, increases at a three-fold rate. These factors make the picker matrix difficult to meet tight power and timing constraints for large schedulers. However, it stands to reason that all-to-all ordering information shouldn't be necessary, especially as scheduler size grows. The conflict resolution logic is only needed when there are resource conflicts — and modern cores have far more resources than their average throughput (instructions per cycle, or IPC) utilizes in traditional benchmarks.

3.2 Age Tracking

In order to quantify the importance of age-based ordering within a picker, we add a pseudo-random picker algorithm in our simulator. In other words, every time multiple instructions are bidding for the same resource on the same cycle, the picker chooses a pseudo-random one. To make our experiments deterministic, the algorithm is actually choosing the instruction most towards the top of the scheduler to win the conflict. After a very short startup phase, the scheduler's out-of-order insertion and removal makes this a sufficiently random choice in terms of relative instruction age. It is important to note that randomly picking, or any picking algorithm for that matter, is not at risk of incorrectness. Only ready instructions are eligible for picking, and eventually the oldest instruction will have to get picked when it is the only ready instruction in the window. Thus, poor picker heuristics cannot cause deadlock.

Figure 8 plots the relative performance difference between the perfect and pseudo-random picker (both the benchmarks and simulator are described in further detail in Section 4). The figure shows that across the benchmark suites scheduler size is a very strong component of age sensitivity. For small schedulers, such as the 16-entry configuration, randomly picking ready instructions only results in a 1% slowdown. A reasonable case could be made for removing age-tracking altogether in a small design such as this. The larger windows, though, show up to 10% average performance loss with a random picker, with larger effects in high-parallelism applications. In scheduler sizes or applications where more ready instructions bid for execution, the odds of randomly choosing a non-critical instruction becomes more likely. With very large schedulers (128 entries) age-criticality can drop somewhat because the perfect-age performance has saturated; however, the random-age performance continues to creep up slowly since its poor picks make it behave like a smaller scheduler. Regardless, considering the evolution of desktop microprocessors towards larger windows and the complexity of tracking a large number of ages, a solution for scalable age tracking is clearly indicated.

3.3 Related Work

Early in the era of out-of-order execution, Butler and Patt [6] studied the efficacy of different picker criteria such as number of dependents, whether the instruction feeds a branch, the dataflow-chain length, and others. They concluded that performance was largely independent of the heuristic used, and simpler was thus better for real designs. Our results from the previous subsection show that less aggressive machines like the kind studied by Butler have little sensitivity to picker heuristic. However, we have also shown that more aggressive machines with larger issue windows need some form of picker ordering.

However, determined industry and academic researchers have attempted to address the monolithic complexity of the picker. The most commonly published approach is to split up the picker and thereby divide and conquer the problem. Partitions can be created by several different heuristics [1, 22, 23, 26, 27], though the most common in industry is by execution resource. The cost of any partitioning technique, however, is the inevitable loss of efficiency, shown in Figure 9 (simulation and benchmark parameters detailed later in Section 4). The baseline curve shows our default machine which has a unified issue queue serving six functional units. The 6-way partitioned curve shows the same machine, except where the total scheduler size has been hard-divided equally among our 6 functional units. Instructions are assigned to partitions based solely on their assigned unit, with a load-balancing algorithm used to steer instructions with multiple entries in their candidate resource

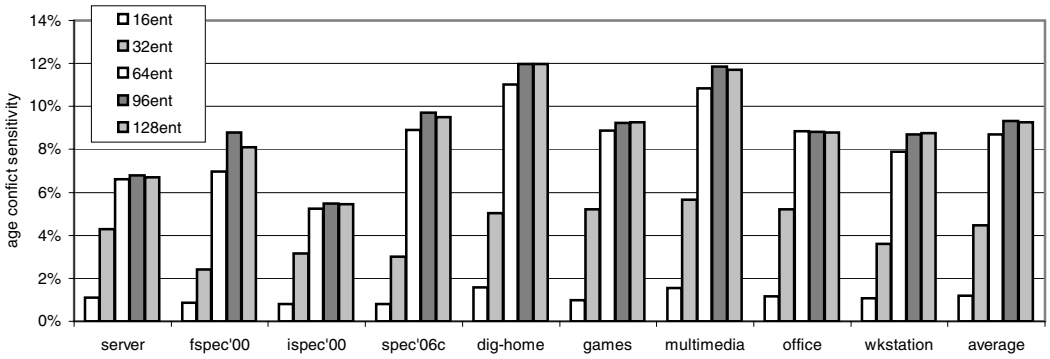

Figure 8: Impact of age-based conflict resolution across scheduler sizes and benchmark categories.

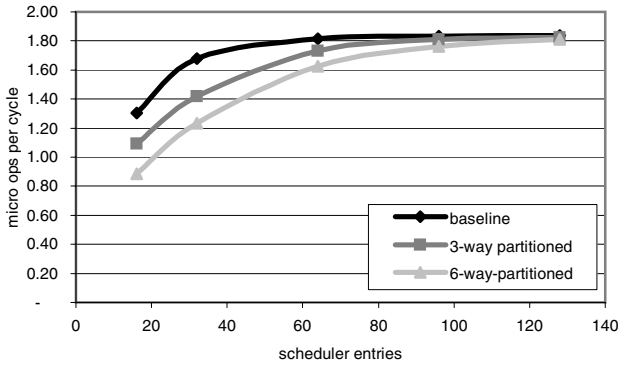

Figure 9: Scheduler scaling curves for a unified scheduler, 3-way partitioned, and 6-way partitioned.

list. The 3-way partitioned curve is similar, except we divide the total scheduler size into three equal segments, where each segment serves two functional units each. The figure shows that partitioned schedulers require several more total entries for an equivalent per-cycle performance against a unified design. For instance, a performance target of 1.60 micro-ops per cycle in our simulator would require about 27 unified entries, 48 three-way partitioned entries, or 60 six-way partitioned entries. Multilevel partitioning techniques [4, 18] are a related approach which resemble cache hierarchies by using a small, fast wait-match buffer backed by larger, slower buffer. Thus the critical scheduler loop can be tighter while allowing far-flung parallelism to be eventually discovered. Importantly, these techniques do not directly address wakeup scalability — we still need a way of communicating operand readiness between partitions.

There has also been academic work in attempting to reduce use of the picker. Select-free instruction scheduling [5] and grandchild scheduling [30] both move the picker into a separate pipeline stage from wakeup. Similarly, cyclone [8] proposes to replace the picker logic with a dataflow prescheduler which places instructions into timed execution queues. In all these works, execution becomes dataflow speculative — the instruction may not be ready when it is picked. Thus a recovery mechanism such as replay must be present in these approaches to insure proper execution. Many academic authors have implicitly addressed pickers in other related areas. Work on critical path discovery and exploitation is especially relevant to our discussion because it has shown that age is not the optimal conflict-resolution heuristic [10, 11, 31]. Criticality prediction, however, is quite complex and largely ineffective at producing speedup.

There have also been several commercial techniques for picker scaling. A generic technique found on many older out-of-order microprocessors is age estimation, sometimes called pseudo-FIFO in reference to cache replacement pseudo-FIFO techniques. Our proposal can also be considered pseudo-FIFO because it estimates age, but we believe it to be a fundamentally different approach than what is likely used in commercial microprocessors.

Another commercial innovation in picker design is the compacting scheduler, disclosed by the DEC Alpha 21264 architects [9, 19], which provides an interesting alternative to expansive picker logic. The concept is simple — always allocate scheduler entries at the top of the scheduler and constantly compact the instructions towards the bottom into the deallocated slots. In this manner, the scheduler stays physically ordered by age despite allowing arbitrary dispatch and deallocation. Thus the age comparison in this design is replaced with a simple priority arbiter (or tree of arbiters), which takes in bids and returns one grant for the bottom-most (oldest) bid. The advantage of the compacting scheduler is that it achieves an age-based pick without just-in-time age comparisons or the inefficiencies created by partitioning. On the negative side, constantly shuffling instructions downward combined with the massive number of write ports creates obvious energy issues, especially for large schedulers.

3.4 Hardware Modifications

Rather than try to maintain the order of every instruction against every other, we instead seek to produce a loose ordering by grouping instructions. We will track which groups are older than other groups, but within a group the age will not be tracked. Figure 10 shows the hardware changes that we need to implement group ordering. The approach uses an age matrix which allocates rows to every scheduler entry as normal, but allocates columns to instruction groups. As later results will show, the number of picker matrix columns needed for group-age tracking can be 60-90% smaller than the baseline with minimal performance impact.

The most minor hardware change is a group counter which could be implemented anywhere from decode down to scheduler allocation. This logic simply assigns a group number to each instruction, changing after a fixed number of instructions. This group number then indexes into a Picker Allocation Table (PAT), somewhat similar to the Wakeup Allocation Table from the previous section. The PAT uses the group number as an index to lookup which age-matrix column this group is assigned to. If no column is assigned to this group yet (first instruction from group), a column is pulled from the Picker Free List and assigned to the instruction and written into the PAT at that index. If the free-list is empty, the allocation must stall until a column becomes free. The reader should note the WAT and PAT are accessed in parallel to the numerous other

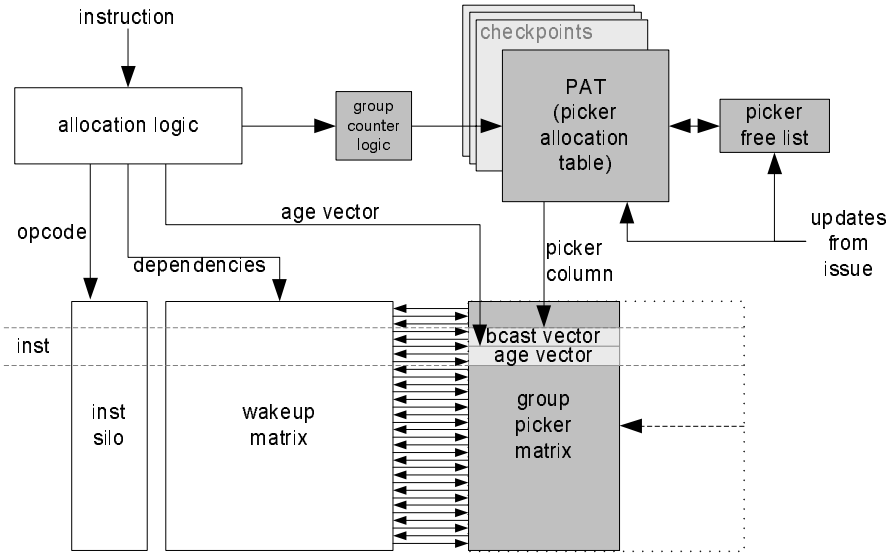

Figure 10: Hardware changes for a faster picker with group-age conflict resolution (not to scale).

checks that occur during the allocate stage(s) of the pipeline, so we do not expect this logic to extend the critical path.

Though we could technically use the group ID (with rollover) as the column ID and skip the translation logic, it is advantageous to allow more live groups than columns. An illustrative example is that column 1 is assigned to group 1, column 2 to group 2, etc., until all columns are filled. The machine then stalls for lack of columns, which is the correct behavior. Then suppose all the instructions in groups other than group 1 issue. Intuitively we should have several free columns to unstall allocation, but without group-to-column translation, the allocator has wrapped around to group 1 and will remained stalled. In other words, the lack of indirection between groups and columns forces columns to be allocated in-order. This is unduly restrictive and causes large performance drops in our simulations. Thus we use a simple group counter and map table to allow columns to allocate out-of-order.

The picker matrix cells are also modified slightly. Figure 11 shows a possible dynamic circuit implementation of a picker matrix cell, with new additions shaded in grey. As with the wakeup matrix diagram in Figure 6, there are several other equally valid ways of implementing this logic; the example shown is only illustrative of how minor the changes are. In the traditional square matrix design, each entry is hard-coded to broadcast conflicts on its transpose column (equal to the row number). For our group picking functionality, however, we require each column to be connected to every column's conflict line. As such, we need to handle two cases: (1) the cell is on a row assigned to the current column, and (2) the cell is on a row assigned to a different column. Thus the circuit shown has two pull-down paths for the two different cases. When the cell is assigned to this column, we need to broadcast the conflict up and down the vertical conflict line. We also need to kill the outgoing grant if the arbiter (described below) indicates a conflict in this column — another instruction in this group has priority. The other case is the traditional case, where we kill the grant if the age bit is set — this group is older than another bidding group. Overall, we have added one SRAM cell indicating if this cell belongs to this column, and three transistors gating the two different behaviors. These changes only lengthen the critical path by only two transistors for the entire matrix since all cells compute their kills in parallel.

However, the picker also needs an arbiter to make a random choice when there is a conflict within a column. In the traditional design this could never occur since only one instruction is assigned per column. There are several ways to choose randomly, but we choose the priority arbiter used in the compacting scheduler. This logic tree is designed specifically to give a fast grant based only on the physical location of the input bids. This is a convenient circuit to use because, though the pick is ignorant of the critical path, the determinism of the heuristic makes hardware debugging tractable. Though this arbiter adds to the critical path along with the two transistors above, we feel the deleterious impact is far less than the dramatic shrinking of the horizontal ready and grant lines.

4. RESULTS

To evaluate our wakeup and picker techniques, we modify our x86 platform simulation infrastructure to model our two proposals, first separately and then together. The simulator thoroughly models a microarchitecture of a hypothetical future microprocessor with accompanying chipset and memory. Key parameters of the model are shown in Table 1. The simulator executes Long Instruction Traces (LITs) which are checkpoints of a complete machine state, including memory, that can be used to initialize an execution-based performance simulator. LITs also include the interrupt injections observed by a real machine executing the application, thus our simulation environment allows us to model user-mode and kernel-mode instructions in the same manner that a real system does. Similar to the SimPoint methodology [15], each LIT runs for a characteristic portion of the application (on average, 6 million instructions) after warming up the caches and branch predictors. LITs are gathered from various categories, elaborated in Table 2 with a total of 604 LITs being studied.

4.1 Wakeup Side Results

In order to determine the efficacy of wakeup subscriptions, we first analyze the number of wakeup columns we need for acceptable performance across a sweep of scheduler sizes. This experiment aims to verify the statistics from Section 2 that few instructions are using broadcast channels at any given time.

Figure 12 shows these results. The first bar is our baseline, an

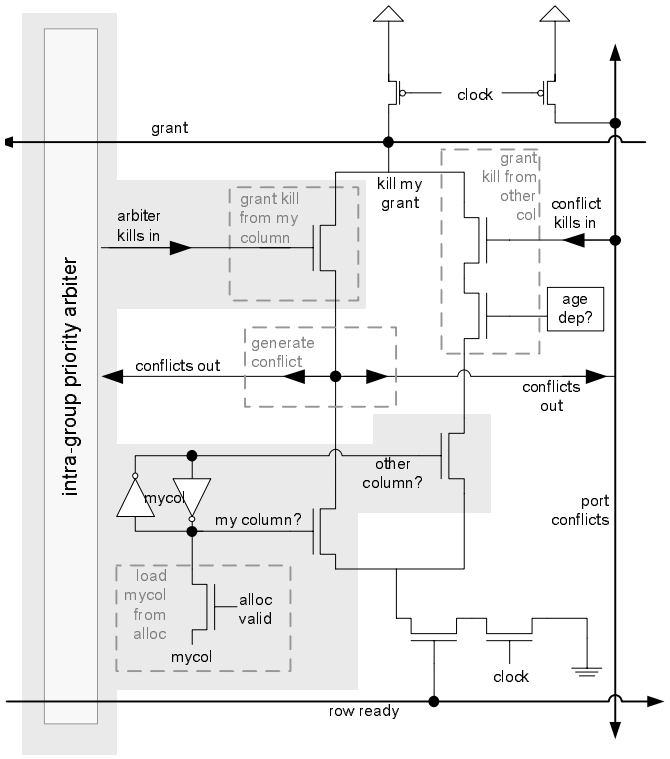

Figure 11: Example picker matrix cell with new hardware shaded in grey.

Parameter	Value
Front End Width	4 wide
Commit Width	4 wide
Execution Units	3 heterogeneous int/FP units [12]
Memory Units	2 load/store units
Reorder Buffer	256 entries
Load Queue	96 entries
Store Queue	64 entries
L1I Cache	32KB, 8 way, 64B line, 4 cycles
L1D Cache	32KB, 8 way, 64B line, 4 cycles
L1 TLB	128 entries, 4 way
L2 Cache	512KB, 8 way, 8 cycles
L2 TLB	512 entries, 4 way
L3 Cache	4096KB, 16 way, 20 cycles
Memory	32GB/s DDR2 timings
Branch History	2048 entries, 4 way
Branch Targets	4096 entries, 8 way

Table 1: Primary parameters for machine simulation model.

Benchmark Class	Example Applications
Server	SpecJBB, TPCC
FSPEC 2000	wupwise, ammp
ISPEC 2000	gzip, gcc
SPEC 2006 candidates	gromacs, mysql
Digital Home	video encode, decode
Games	shooters, realtime strategy
Multimedia	photo filter, raytracer
Office	word processor, spreadsheet
Productivity	file compression, doc rendering
Workstation	CAD, compiler

Table 2: Benchmark suites used for performance analysis.

unaltered simulation using our default parameters. The remaining three bars use our wakeup subscription algorithm with 16, 12, and 8 columns respectively. Per-cycle slowdown from the baseline are shown at the top of each bar. It is clear we are taking advantage of the low demand for broadcast-wakeup communication channels shown in Figure 4, and thus very few columns are needed even with very large schedulers. A wakeup matrix of only 12 columns produces less than a percent slowdown regardless of scheduler size. Larger schedulers need more columns to maintain a constant slowdown, but only slightly more are needed. Just as the marginal gain of additional scheduler entries diminishes, so it is with additional wakeup columns.

As each bar in Figure 12 is an average of approximately 600 benchmarks, much detail is abstracted away. So we break down the 64-entry bars and plot the slowdown for each benchmark suite in Figure 13. Here we can see a great deal of variation, especially in the 8 wakeup column case, being hidden by the large averages in the previous figure. Digital home benchmarks, with their high average IPC (or micro-ops per cycle in our case), are highly susceptible to anything that decreases the effective issue window size. FSPEC, which also has a high IPC, has relatively few producer-consumer relationships within the scheduler, thus making conservation of wakeup columns less relevant to this suite.

Interestingly, the FSPEC00 suite speeds up from the use of wakeup subscriptions. On the face this is counter-intuitive as our algorithm can only add stalls, not remove them. However, sometimes a larger scheduler combined with an idle and eager load unit will allow more errant loads to be issued, tying up important memory subsystem resources (cache ports, miss status handling registers, etc.) even after the mispredicted branch is exposed. Usually this slowdown effect is most apparent with small changes in scheduler size, as large strides allow enough additional parallelism to outweigh the errant load effect. This effect is at work in several benchmarks across our suite, mostly concentrated in ISPEC00 and FSPEC00. Wakeup subscriptions make the scheduler appear slightly smaller when columns are in high demand, thus these benchmarks profit from errant loads being excluded from the scheduler.

We also observe the effect of column reallocation on performance. Often with a free-list approach, microarchitectural resources are not available for reallocation on the cycle after they are freed. Signal propagation delay, combined with careful bookkeeping to avoid losing resources, means it might be a few cycles before this resource is ready for allocation. Figure 14 shows the sensitivity of our algorithm to delays between a wakeup column becoming free (producer and/or consumers leave scheduler) and when it is available for allocation to an incoming instruction. We evaluate this on a 64-entry scheduler over a range of 0 to 3 reallocation cycles for various column counts. Figure 14 shows that performance can drop noticeably if the number of reallocation cycles grows too large with too few wakeup columns. As increasing the reallocation delay effectively reduces the effective number of wakeup columns, we can compensate for additional delay with more columns. For example, the data shows that 12 wakeup columns with 2 reallocation cycles is equal performance with 16 columns with 3 reallocation cycles.

These small IPC losses shown are offsets against the reduced critical path distance through the wakeup matrix. The critical path for a wakeup matrix is a grant to the top-most entry, which sends a broadcast on the first column down to a consumer at the bottom-most entry, which then becomes ready and flips its row-ready line which heads back to the picker. In other words it is a line around three sides of the matrix perimeter, shown by the shaded line in Figure 3.

To quantify this value, we count the number of cell hops on the critical path through the matrix with and without wakeup subscriptions. This is clearly a crude metric, but the custom circuit, floorplan, and process variations affecting scheduler designs make more specific numbers hazardous. Figure 15 shows the result of

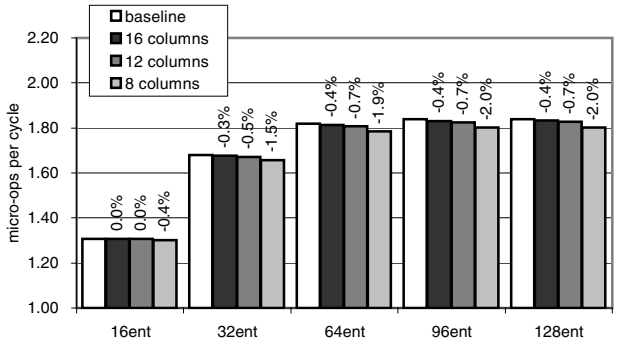

Figure 12: Performance impact of various wakeup column counts.

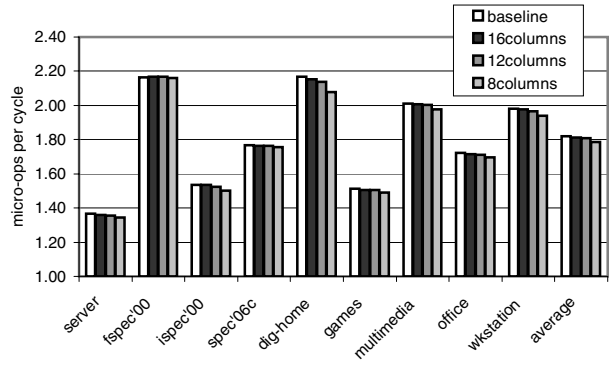

Figure 13: Per suite performance slowdown for 64-entry scheduler with 16 wakeup columns.

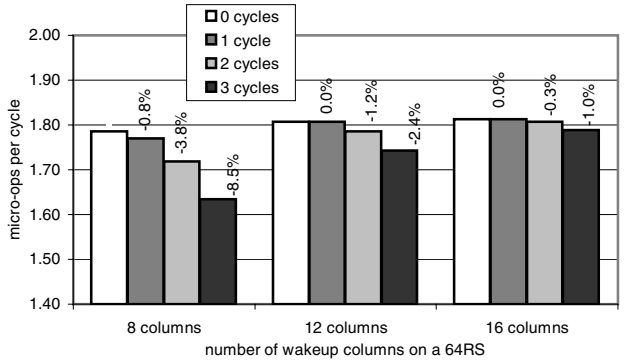

Figure 14: Sensitivity to wakeup column reallocation delay for a 64-entry scheduler.

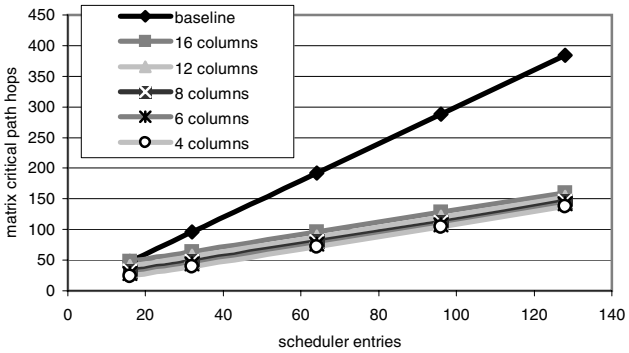

Figure 15: Critical path hops through wakeup matrix for various heights and widths.

this critical path analysis for different scheduler sizes and wakeup matrix widths. The top line is the baseline machine with a traditional square wakeup matrix. The unscalability of the square design is clear — the critical path increases proportionately with the number of entries. It is difficult to justify this delay increase even with the IPC improvements that larger schedulers afford. The remaining lines show the critical paths of a rectangular matrix with various widths. It is evident that these delays scale far better than the traditional square matrix design. This allows us to design a large scheduler with the critical path delay of a much smaller one. For instance, a 64-entry 12-column configuration has similar delay to that of a traditional 32x32 configuration, yet it has a 7.5% higher performance. Similarly a 32-entry 8-column configuration has a similar delay to a traditional 16x16 but 26.5% higher performance.

4.2 Picker Side Results

The first step in evaluating the group picking technique is to determine how to divide up our groups. Groups are the quantum that will be ordered in our picker. If the groups are too big, the selection will degrade to the random picker, which we have seen is not acceptable. If the groups are too small, then we will need more columns to track all the instructions in the taller schedulers. Our first intuition was to divide at basic block boundaries, but the high variance in block size created fragmentation in our picker design, dramatically reducing performance. So we instead cut groups after a constant number of instructions, regardless of control boundaries.

To determine the optimal number of instructions per group, we need to observe the relationship between the number of picker columns and the number of instructions per group. As an illustration we use a 64-entry scheduler, though other scheduler sizes follow similarly. Figure 16 (top) plots the performance of various group-size and column-count configurations. This simulator has a baseline wakeup matrix, but alters the picker so that groups are ordered but instructions within a group are not. Figure 16 (bottom) shows this same data as a surface map so we can more clearly see the sweet spot for number of instructions per group. The reader should note this quantity is essentially a free-choice; that is, the size chosen only affects the size of the group counter, a trivial concern. The number of columns, however, is far more important. The objective of group-based picking is to reduce the number of picker columns while maintaining acceptable performance. The map in Figure 16 (bottom) shows us that we can use fewer columns if we choose the proper number of instructions per group. Thus we should choose the group size where the saddle of the white-shaded (highest performance) area is, because that is where we need the fewest number of columns for a given performance level.

Interestingly, the optimal number of instructions per group is around 10-14 for all scheduler sizes in our simulations, thus we choose 12 instructions per group for all our performance experiments. Figure 17 shows these performance results across a variety of scheduler sizes. The first bar in each group is the baseline picker, a configuration with perfect age resolution. The remainder of the bars show group picking with a reduced number of columns. Configurations which do not allow the full capacity of the scheduler to be reached are omitted. For instance, at 12 instructions per group, 128 scheduler entries requires at least 11 columns to track all entries. Thus we do not include smaller column counts for the taller schedulers.

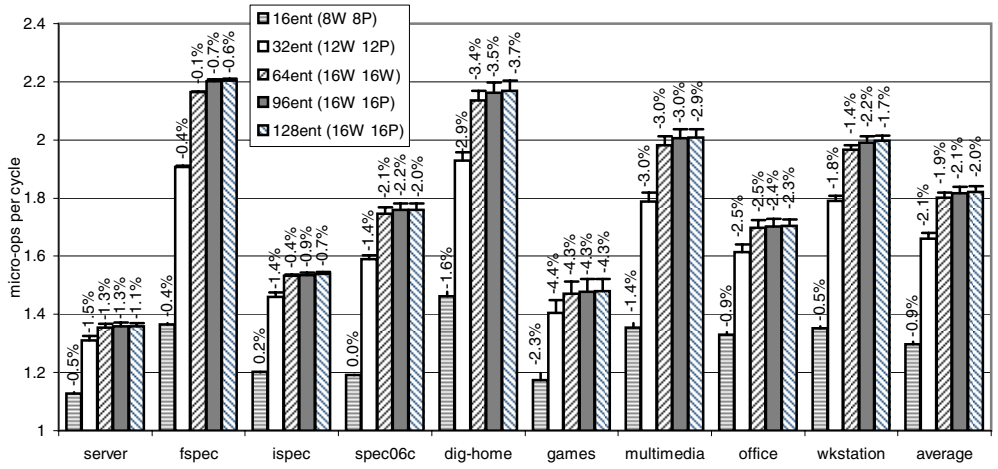

Figure 18: Performance of wakeup and picker techniques combined for different benchmark suites.

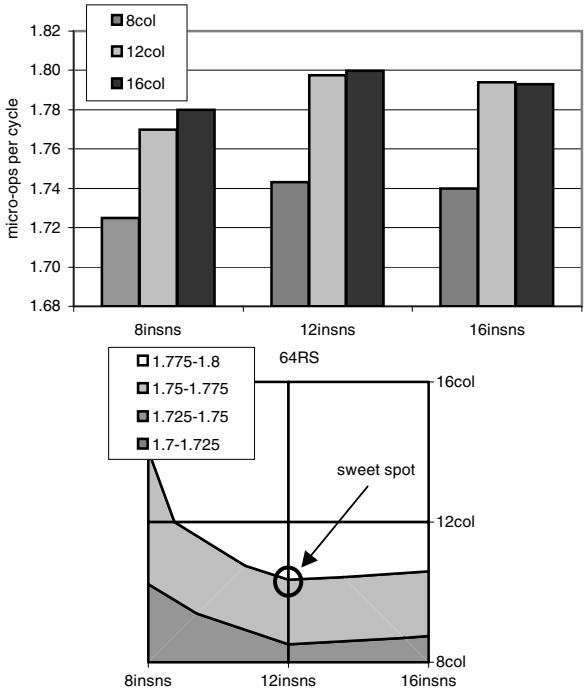

Figure 16: Chart (top) and surface map (bottom) plotting the performance (micro-ops per cycle) at different column counts (col) and micro-ops per group (insns) for a 64-entry scheduler.

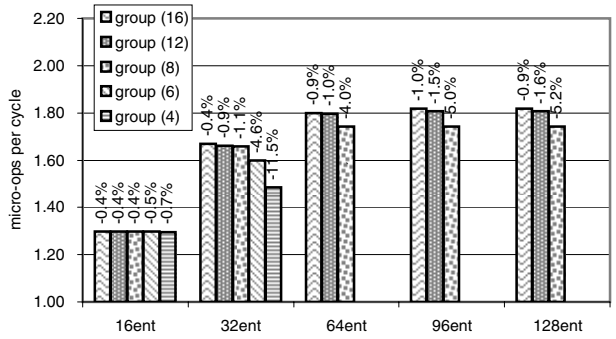

Figure 17: Performance of group picking across a range of scheduler sizes. Group size is 12 instructions in all group-picking configurations.

We can see from the results that group picking is highly effective at reducing the amount of age tracking needed, though the ideal number of columns is still related to scheduler size. A 16-entry scheduler loses only 0.7% per cycle performance by tracking the relative age of only 4 groups of instructions rather than all 16 individual instructions. The 128 entry scheduler loses only 1% by tracking 16 group ages rather than all 128. For brevity we omit an analysis of picker column reallocation delay, but the results are quite similar to the wakeup side (see Figure 14): additional delay incurs marginal performance costs but can be compensated for with an extra couple of columns. Even with these additional columns, group picking reduces the size and latency of the matrix significiantly from the baseline picker design. This latency reduction might allow looser timing in the wakeup phase.

Alternatively, scheduler height can be added without significant impact on the critical path through the picker. This path is similar to the wakeup side path — a line across the bottom, up the farthest side, and back across the top. Thus we can use the same matrix delay estimates from in Figure 15 to track the number of hops through the picker matrix. As with the wakeup matrix, cell hop counting is a crude metric but is agnostic to process, floorplan, and circuit variations. Given the need for only 8-16 columns, the data in Figure 15 indicates a similar improvement in delay on the picker side as on the wakeup. For instance, a 64-entry 16-column picker has the same number of hops as a 32-entry square picker, but has 7.3% higher performance.

4.3 Combined Results

We have shown so far, separately, that the wakeup and picker enhancements allow significant improvement in the scalability of the two scheduler segments. In our final analysis, we combine the techniques into a single scheduler configuration and compare against our baseline matrix scheduler. Figure 18 plots the combined performance numbers for various scheduler sizes. Error bars and data labels indicate performance loss from our baseline scheduler with complete wakeup and picker matrices, and the legend indicates the number of wakeup and picker columns used for each scheduler

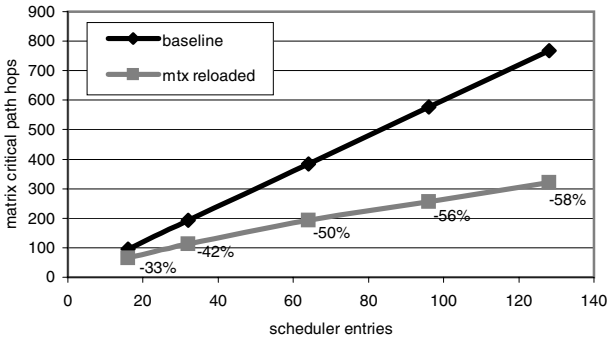

Figure 19: Critical path hops through total baseline matrix scheduler and thin matrix scheduler for various scheduler heights and widths.

height. The graph shows that most configurations have between 1 and 2% average performance loss from the baseline — the equivalent of cutting the baseline scheduler by few entries.

However, Figure 19 shows that the wakeup and picker thinning techniques decrease the scheduler loop latency much more than dropping a few entries would. Our latency estimation metric, cell hop count, decreases from 33% with a 16-entry scheduler to 58% with a 128-entry scheduler. Interestingly, the path delays are nearly identical for a baseline 32-entry scheduler and a 64-entry scheduler with thin wakeup and picker matrices, a trade producing an average 8.3% performance gain for similar timings. Even accounting for slightly more complex cells in both matrices, the efficiency improvement is clear.

5. CONCLUSION

Our goal in this work is to provide greater published understanding of matrix schedulers, to reveal their design advantages and scalability weaknesses, and improve on the basic approach through simple indirection techniques. On the wakeup side, we have shown that the all-to-all tag broadcast in a traditional matrix is over-design: only a handful of tags need broadcast, even in large schedulers. By conservatively subscribing to broadcast channels (wakeup columns) we can reduce the size and wire-delay through the matrix significantly. An additional structure is required to map registers to columns, but it likely amortizes its own cost by reducing the matrix size and eliding the need for the large bank of traditional allocation comparators. Similarly, we have shown that the picker matrix is over-designed for providing full age resolution. Indeed, some level of age tracking is needed, but ordering small groups of instructions appears to be adequate. IPC drops by less than a percent if the number of groups in-flight (picker columns) is kept reasonable. By reducing the width of the both matrices by 60-90%, the total critical path latency is greatly shortened and possible frequency and power improvements can be realized.

It is important to note that, though the machine chosen for our simulations represents a reasonable future core, both of these approaches are already useful in current microprocessors. We do not require large out-of-order monolithic pipelines to reap the benefits of more scalable issue windows. If architects do wish to increase machine width and depth, though, we hope that techniques such as ours assure that schedulers do not become scalability bottlenecks. In the opinion of the authors, a good use for such width and depth is simultaneous multithreading, for which a large scheduler is clearly useful. A study of our wakeup and picker techniques with SMT workloads is on-going work.

6. REFERENCES

[1] AMD software optimization guide for AMD64 processors, pub 25112, rev 3.06, www.amd.com.

[2] E. Borch, E. Tune, E. Manne, S. Emer, Loose loops sink chips, in *Proceedings of HPCA-8*, Feb. 2002.

[3] A. Bracy, A. Prahlad, P. Roth, Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth, in *Proceedings of MICRO-37*, 2005.

[4] E. Brekelbaum, J. Rupley, C. Wilkerson, B. Black, Hierarchal scheduling windows, in *Proceedings of MICRO-35*, 2002.

[5] M. Brown, J. Stark, Y. Patt, Select-free instruction scheduling logic, in *Proceedings of MICRO-34*, 2001.

[6] M. Butler, Y. Patt, An investigation of the performance of various dynamic scheduling techniques, in *Proceedings of MICRO-25*, 1992.

[7] D. Ernst, T. Austin, Efficient dynamic scheduling through tag elimination, in *Proceedings of ISCA-29*, 2002.

[8] D. Ernst, A. Hamel, T. Austin, Cyclone: a broadcast free dynamic instruction scheduler with selective replay, in *Proceedings of ISCA-30*, 2003.

[9] J. Farrell, T. Fischer, Issue logic for a 600-Mhz out-of-order execution microprocessor, in *IEEE Journal of Solid State Circuits*, Vol. 33, No. 5, May 1998.

[10] B. Fields, S. Rubin, R. Bodik, Focusing processor policies via critical-path prediction, in *Proceedings of ISCA-28*, 2001.

[11] B. Fields, R. Bodik, M. Hill, Slack: maximizing performance under technological constraints, in *Proceedings of ISCA-29*, 2002.

[12] A. Fog, The microarchitecture of Intel and AMD CPUs, www.agner.org/optimize/microarchitecture.pdf, Aug 13 2006.

[13] A. Gonzales, M. Valero, Virtual Physical Registers, in *Proceedings of HPCA-4*, 1998.

[14] M. Goshima, K. Nishino, Y. Nakashima, S. Mori, S. Tomita, A high-speed dynamic instruction scheduling scheme for superscalar processors, in *Proceedings of MICRO-34*, Dec 2001.

[15] G. Hamerly, E. Perelman, J. Lau, B. Calder, SimPoint 3.0: faster and more flexible program analysis, *Journal of Instruction Level Parallelism*, Sep 2005.

[16] I. Kim, M. Lipasti, Half-price architecture, in *Proceedings of ISCA-30*, 2003.

[17] K. Krewell, Intel Looks to Core for Success, in *Microprocessor Report*, Mar 27 2006.

[18] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, E. Rotenberg, A large, fast instruction window for tolerating cache misses, in *Proceedings of ISCA-29*, 2002.

[19] D. Leibholz, R. Razdan, The Alpha 21264: a 500MHz out-of-order execution microprocessor, in *Proceedings of IEEE Compcon*, 1997.

[20] E. Marques, C. Kirner, Design of the matching unit of a massively parallel dataflow computing system, in *Proceedings of IEEE Conference on Massively Parallel Computing Systems*, May 1994.

[21] P. Michaud, A. Seznec, Data-flow prescheduling for large instruction windows in out-of-order processors, in *Proceedings of HPCA-7*, 2001.

[22] S. Palacharla, N. Jouppi, J. Smith, Complexity-effective superscalar processors, in *Proceedings of ISCA-24*, 1997.

[23] J. Parcerisa, J. Sahuquillo, A. Gonzlez, J. Duato, On-chip interconnects and instruction steering schemes for clustered microarchitectures, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 16, No. 2, Feb 2005.

[24] P. Sassone, D. Wills, Dynamic strands: collapsing speculative dependence chains for reducing pipeline communication, in *Proceedings of MICRO-37*, 2005.

[25] P. Sassone, D. Wills, G. Loh, Static strands: safely collapsing dependence chains for increasing embedded power efficiency, in *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[26] J. Shen, M. Lipasti, *Modern Processor Design*, McGraw Hill, 2003.

[27] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, J. Joyner, "POWER5 system microarchitecture," IBM Journal of Research and Development, Vol 49, No. 4/5, July 2005.

[28] J. Smith, A. Pleszkun, Implementing precise interrupts in pipelined processors, *Proceedings of Computer Architecture*, 1985.

[29] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, M. Upton, Continual flow pipelines, in *Proceedings of ASPLOS-11*, Oct 2004.

[30] J. Stark, M. Brown, Y. Patt, On pipelining dynamic instruction scheduling logic, in *Proceedings of MICRO-33*, 2000.

[31] E. Tune, D. Liang, D. Tullsen, B. Calder, Dynamic prediction of critical path instructions, in *Proceedings of HPCA-7*, 2001.